What is meant by “Real-Time” Programming?

- It does not mean “really, really fast”
- It means that there are time constraints that the application attempts to achieve

Soft real-time: “it would be nice if X happens within Y seconds”
Hard real-time: “it would be disastrous if X doesn't happen within Y seconds”

The main challenge to real-time programming is **non-determinism**

- we don't know what will happen
- we don't know when it will happen

What are the sources of non-determinism (uncertainty) in a C program?

- Scheduling of other processes and threads
- Interrupts with higher priority
- Cache misses (to some extent)
- Virtual memory paging
- Response times to interrupts
- System calls

All of these are dealt with by the **operating system**: no matter what you do to your C code, you can't control those sources of non-determinism

OS usually tries to improve the common case; in RT, we need to worry about the worst case

**POSIX Real-time Support**

- FIFO scheduling: fixed-priority, preemptive (still susceptible to priority inversion)
- Virtual Memory locking: lock entire address space of program into virtual memory (no paging)

**RTLinux**

http://en.wikipedia.org/wiki/RTLinux
http://www.soe.ucsc.edu/~sbrandt/courses/Winter00/290S/rtlinux.pdf

- Want to be able to run “regular” processes along with RT processes
- The standard Linux kernel is itself a task – it is the default “idle” task, meaning that it will only run if there are no RT processes to be executed
- The standard Linux task cannot disable interrupts or prevent itself from being preempted
- RT processes always have priority... if a standard Linux process is starved, too bad
- RT process never blocks on a resource it's sharing with a standard Linux process
- no malloc in RT processes, just static memory

**RTOS scheduling algorithms**

The OS needs to schedule RT threads in such a way to ensure that they meet their deadlines
A needs to run 33 times/second, or start/finish every 30ms, and takes 10ms to run
B needs to run 25 times/second, or start/finish every 40ms, and takes 15ms to run
C needs to run 20 times/second, or start/finish every 50ms, and takes 5ms to run

Are these schedulable?

\[ Ci = \text{CPU time for process } i \]
\[ Pi = \text{period for process } i \]

it's schedulable if sum of \( Ci/Pi \) is less than or equal to 1
in this case, we have \( 10/33 + 15/40 + 5/50 = 0.808 \) so we're okay
what if we added another process D that runs at 40 times/second (finishes every 25ms) and takes 5ms to run? Now we add 5/25 and we're over 1, so this can't be scheduled!

Rate Monotonic Scheduling (static real-time scheduling algorithm)

processes must meet following conditions:
1. the process must complete within its period (obviously can't take longer than 33ms to do the work if you need to do it 30 times/second!)
2. process is not dependent on any others (to avoid waits)
3. needs the same amount of CPU time on each burst (or, can take up that amount of time)
4. process preemption occurs instantaneously (is this realistic?)

Each RMS process is assigned a priority based on the frequency at which it must finish:

scheduler always runs highest priority ready process, preempting the running one if needed

A needs to run 33 times/second, or start/finish every 30ms, its priority is 33...
B needs to run 25 times/second, or start/finish every 40ms, its priority is 25...
C needs to run 20 times/second, or start/finish every 50ms, its priority is 20

Earliest Deadline First Scheduling (dynamic real-time)

doesn't require processes to be periodic or to run in fixed bursts
when a process needs CPU time, it specifies its deadline
scheduler selects one with earliest deadline

limitations aside, they usually provide more or less the same scheduling, but there are cases when EDF succeeds whereas RMS fails

A: period = 30, time = 15
B: period = 40, time = 15
C: period = 50, time = 5

In this case, RMS will cause C to miss its deadline (because A and B “starve” it) whereas there would be no problem with EDF

Real-time Java

Why is real-time programming particularly hard in Java?
• dynamic class loading
• garbage collection
• JIT compilation
• thread scheduling

How can you address these issues without any special support?
• Scheduling: there's no guarantee for thread priorities (JVM not required to enforce priorities!), so you don't know when your thread will run; solution is to partition the application into high-priority app and low-priority app, then run them in separate VMs with separate priorities, and let the OS deal with it
• Class loading: a class definition is loaded from disk into memory the first time the class is used; that can take a variable amount of time (around 10ms); need to pre-load all classes before getting into the RT section of code
• Garbage collection: GC runs whenever heap is getting full... or whenever it feels like it; never sure how long it will take (100s of ms?), and all other threads have to wait for it to finish; you could try to reduce GC by reusing objects so that you don't allocate more memory, but you don't have control over underlying libraries
• JIT compilation: you don't know when the compiler will decide to JIT the code, or how long it will take; you could try calling methods multiple times before getting into RT section of code, to make it more likely that they will already have been compiled to native code

RTSJ: real-time specification for Java

Scheduling:
With the RTSJ, true priorities and a fixed-priority preemptive scheduler with priority-inheritance support is required for RT threads. This scheduling approach ensures that the highest-priority active thread will always be executing and it continues to execute until it voluntarily releases the CPU or is preempted by a higher-priority thread. **Priority inheritance** ensures that **priority inversion** is avoided when a higher-priority thread needs a resource held by a lower-priority thread.

Memory management
Objects allocated in the immortal memory area are accessible to all threads and are never collected. **Scoped memory** is used only while a process works within a particular section, or scope, of the program such as in a method. Objects are automatically destroyed when the process leaves the scope. This is a useful feature akin to garbage collection in that discrete creation and deletion is not required as in the immortal memory case - but the process must be sure to exit the scope to ensure memory is reaped. Neither immortal nor scoped are GC’d

Threads
RT systems typically use NHRTs (no heap realtime... can't access heap) with high priorities for tasks with the tightest latency requirements (not blocked when GC runs), **RealtimeThread**s for tasks with latency requirements that can be accommodated by a garbage collector, and regular Java threads for everything else. 28 levels of priority that are strictly enforced
Clocks
higher resolution than normal VM

Using RTSJ: memory

Assume you have a time-critical thread and a non-time critical thread.... even if time-critical has higher priority, the other can interrupt it if it does something that triggers the GC

RTSJ allows for different memory areas that are not GC’d

ImmortalMemory: there is only one instance so you access it through the singleton method “instance”. You then “enter” that area and pass it a Runnable object... anytime you call “new”, the object is put into that area and is never GC’d

```
ImmortalMemory.instance().enter(new Runnable() { public void run() { … } });
```
NOTE: this doesn't create a new Thread (to run in parallel), it just runs the “run” method.

ScopedMemory: you can create an instance and size it... then you enter.... any objects created are put into that area and aren't GC'd until the run method finishes

Scheduling

Schedulable interface extends Runnable...

Schedulable object has a release profile (when it's ready for execution), processing cost (how long it takes to run), deadline, etc.
"Release" is when the schedulable object (RT thread) becomes ready for execution
ReleaseParameters include cost (amount of cpu time needed each release) and deadline (relative time at which release must finish), as well as overrun and miss handlers
PeriodicParameters extends ReleaseParameters and lets you specify start time and period

schedulable object is started, then waits to be released (or may be released immediately)
It does its thing, then waits to be released again (completion time)
May be time-triggered (periodic), irregular but with minimum interval (sporadic), or neither (aperiodic)
Scheduling Parameter is used to determine which object is most eligible for scheduling (high priority value means more likely to be executed... 28 different levels)

Priority Inversion: task L has resource R; task H wants R so it blocks; task M runs and prohibits L from running, prohibiting it from releasing R, prohibiting H from running... so M now has higher priority than H (this happened on the Mars Pathfinder)
Priority Inheritance: increase the priority of L to that of H, so that L can release R and allow H to run before M

MemoryParameters: how much memory you expect the object to use, and which areas it will use